AFRL-RI-RS-TR-2015-188

CREMA

ASSURED INFORMATION SECURITY, INC

AUGUST 2015

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Defense Advanced Research Projects Agency Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2015-188 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
TANYA M. MACRINA
Work Unit Manager

/ S /
WARREN H. DEBANY, JR
Technical Advisor, Information
Exploitation and Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
AUG 2015	FINAL TECHNICAL REPORT	SEP 2014 – MAR 2015

4. TITLE AND SUBTITLE

CREMA

5a. CONTRACT NUMBER
FA8750-12-D-0002/0008

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Karen Reilly, Jacob Torrey, Jared Frank, Trent Brunson

5d. PROJECT NUMBER
ACTC

5e. TASK NUMBER
RE

5f. WORK UNIT NUMBER
MA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Assured Information Security, Inc.
153 Brooks Road
Rome NY 13441

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGA
525 Brooks Road
Rome NY 13441-4505

DARPA/I20
675 N. Randolph Street
Arlington, VA 22203-1714

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2015-188

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. DARPA DISTAR Case# 24560
Date Cleared: 15 May 2015

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Crema is a programming language and restricted execution environment of sub-Turing power for building a provably-secure and intent-driven programming language. By restricting the computational expressiveness of programs to a set of minimal requirements, *weird machines* - the unintended execution environments created when data is not explicitly handled or formally parsed - can be eliminated, and programs will only execute according to their author's intentions. The effort estimated and compared the respective sizes of verification tasks for the Qmail SMTP parsing code fragments when executed natively vs in Crema---using LLVM and KLEE. Research was also accomplished to understand the application of the same principles to the verification of reference monitors

15. SUBJECT TERMS

Crema sub-Turing Language, Compiler, Verification, Symbolic Execution Paths

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**TANYA M. MACRINA**
U	U	U	UU	29	19b. TELEPHONE NUMBER *(Include area code)* **N/A**

Table of Contents

List of Figures

List of Tables

1.0 SUMMARY

Crema was a seedling effort that explored the sub-Turing complete (TC) programming languages and execution environments for building a provably-secure and intent-driven programming language. By restricting the computational expressiveness of programs to a set of minimal requirements, *weird machines* - the unintended execution environments created when data is not explicitly handled or formally parsed - can be eliminated, and programs will only execute according to their author's intentions. This technical report follows three main themes. First, we describe the path explosion problem that formal verification processes face and explain how a less expressive computational model can reduce the number of symbolic execution paths dramatically. Following this discussion, the technical details of the Crema language and its development will be described, and finally, we demonstrate Crema's effectiveness in reducing the execution state-space of parts of the *qmail* parsing code using KLEE.

Crema is a general-purpose sub-TC language developed as a C++ front end to the Low Level Virtual Machine (LLVM) compiler suite. A lexer and parser take the input program, validate it against the formal grammar, and check for semantic correctness. What makes Crema different from other languages is that additional checks are made to prevent unbounded recursive function and co-recursive functions, where one function calls another and subsequently calls back to the original. After the semantic analysis pass is completed the abstract syntax tree (AST) is converted to LLVM bytecode, and the output is linked with the developed C standard library (stdlib) using Clang to take advantage of the LLVM compiler, linker, and optimization passes. This design gives software developers the option to revise portions of LLVM-compatible code using Crema, rather than needing to completely redesign and rewrite the entirety of their project.

The results of the Crema research on sub-Turing computations inspired a new model that should be easier to reason about and is more intuitive. Assuming finite memory, a processor environment is imagined in which all jumps or branches can only go to higher memory addresses. This prevents infinite looping and will guarantee termination and create a strict upper bound on state space, which are the remaining branches in the program's memory. This model is enhanced by adding a just-in-time (JIT) loop unroller to the program loader, where it is unrolled into memory *n* times. As long as the loop is sub-Turing, this process allows for the execution of Walther recursive algorithms - a method used to determine whether a loop will definitely terminate given finite inputs.

The Crema team chose KLEE to check programs for their correctness. KLEE was highlighted as an obvious choice because of its open-source nature and support for the LLVM intermediate Representation (IR) bytecode format. Our experiments with KLEE focused on realizing the dual use of Creama and KLEE and measuring Crema's ability to reduce the state space of a program that requires verification. This research found that KLEE can support LLVM IR programs as long as an intrinsic function instructs KLEE where to explore. Since Crema can link with C programs, existing intrinsic functions may be reused. Using KLEE's default three-minute time out limit to prevent path explosion in state space, we compared analogous C and Crema programs. Our results are able to show that KLEE can safely run entire Crema programs without timing out because of Crema's ability to create a smaller state-space.

The Crema team addressed two specific problems in this effort that can be phased in two fundamental questions. Can a restricted computational model eliminate weird machines? And can it systematically reduce the state space to a point where verifying the security of large code bases is now possible? Crema was created to answer these questions according to the main beliefs established by the field of language-theoretic security (LangSec).

2.0 INTRODUCTION

2.1 Problem Statement

A centerpiece of computer security research is the identification and study of exploitation. Exploits are the innovative methods used to leverage weaknesses in computing resources to draw unexpected behavior or perform unintended computations on the system. A classic example of this is to use a buffer overflow to gain access to memory that is allocated for a program and gain control of the function return address in order to redirect computation [1]. For many years, computer security researchers and experts have explored the different ways in which programming languages and their libraries can be used to give unauthorized privileges through creative means. At the same time, software makers have attempted to resolve common security vulnerabilities for many years, so present day exploits are much more sophisticated and do not involve taking advantage of just one buffer overflow or software bug, but rather chain together a series of exploits to drive the program into an unintended state [2].

The discussion of exploitations in software naturally leads to a question of how secure and trustworthy the computer languages themselves are. There is an asymmetry in the amount of work required to formally or symbolically prove whether software is vulnerable or completely trustworthy. Vulnerability proofs may be carried out with a single, simple demonstration, whereas proving and verifying that a program is completely secure requires the anticipation of all possible execution paths, which grows exponentially with the program size. This is commonly referred to as the path explosion problem or the state space explosion problem, and it prohibits or largely restricts the effectiveness and practicality of symbolic execution for most programs that require input.

One of the fundamental notions from LangSec is that the root cause of the majority of security vulnerabilities in input-handling code is the lack of a formal language that specifies valid or expected inputs. Since crafted input data is capable of driving unexpected computation within common software, the prevailing idea is that input validation is similar to program verification, aimed at precluding inputs from driving their handling code into unexpected state through unexpected computation [3]. This verification can only be achieved if the input-parsing code is constructed to reject any inputs outside of a formal-language definition (grammar) of valid or expected inputs. Using such a language would allow programmers to build software components that are constructed as verifiable recognizers [4].

Sometimes software makers may adopt a standard protocol for processing input, where invalid inputs are considered but still expected. In this case, unless *all* such inputs are formally specified as a part of the formal language of input, the rewriting is often co-opted by attackers [5]. To this end, additional design effort and care must be spent on the input-parsing routines. Formal

verification and extensive testing of these routines is an important step toward security assurance, and as such, must be simplified as much as possible.

2.2 Weird Instructions and Weird Machines

Computer programs typically interface with and require input from unknown users. Generally, the programmer must bestow a set of rules for the expected input in her code and properly handle unexpected occurrences. This creates a situation in which the creator draws up a contract of trust that the user is not required to sign, nor does the user suffer any consequences for violating or extending the purpose of the program. Well-crafted input from the user of a vulnerable program can be manipulated to give weird instructions to the machine and put it into an unintended state. This type of data manipulation often requires a multi-step process that can be spread throughout the execution timeline of a program to achieve a single goal [6]. A program in which multiple weird instructions exists creates an execution engine or a computational model that referred to as a weird machine - an ad hoc, emergent virtual machine (VM) that converts input data into execution flow with unexpected states and state transitions [7].

With the use of weird machines, LangSec provides a formalized view of software exploitation. Return-oriented programming (ROP) is an example where a weird machine is constructed by mining an existing code base such as *glibc* for executable *gadgets* ending in RET instructions. In addition to the memory-corrupting bug that overwrites the stack with this program, the chain of faked stack frames of the exploit payload acts as its program and executes on the emergent machine composed of these gadgets. On a closer look, the addresses of these gadgets in memory form assembly-level bytecodes (op-codes) that drive execution of the weird machine to unintended states in the control flow graph (CFG). This construct has been described as early as 2001 by Refs. [8], [9], and its expressive power has been definitively described in terms of Turing-completeness in Reference [10], finally updating the academic security's threat model from the narrowly-defined "malicious code" to the broader "malicious computation." (See [11].)

2.3 Turing Completeness and Security Vulnerabilities

The argument that motivates the work of Crema is that programs often have too much computational privilege that lies dormant until an attacker discovers and exposes it [2]. LangSec highlights the risks involved with parsing input into a program's internal type system and demonstrates how a poorly designed or implemented parser creates a risk of unintended computations. The theory goes further and calls for input languages to be as restricted as possible, and in Reference [4], for programmers to utilize the minimum amount of computational expressiveness as needed.

Crema laid the foundations for useful sub-Turing programming languages. Currently, programming languages aim to be TC to provide the programmer with as much power and flexibility as possible. Turing-completeness, however, prevents complete formal verification of security properties and termination analysis of large programs due to state space explosion and the undecidable nature of the Halting problem - a significant security and reliability concern. In recent research, the case has been made that TC languages are more powerful than most programs require, and that extra power holds the risks of compromise [12]. Even with a strict

input grammar and a well-designed parser, implementation flaws can still undermine the security of the overall program. Without fast and effective program analysis tools, those implementation flaws can persist into the production application.

The Crema effort researched the methodologies required to develop general-purpose sub-Turing programs and provided the roadmap for automated formal language analysis and reduction from a subset of TC programs to sub-TC programs. By laying these foundations, Crema supports a major programming paradigm shift towards provably secure, verifiable and still usable sub-Turing languages, shifting the balance of power away from the cyberspace attacker. Crema collapses the grammar complexity to the least powerful language class necessary to compute the program's functionality (down in terms of the Chomsky hierarchy) and allows software to be verified before and during execution to prove that the program has not been compromised [13]. This approach is a significant improvement over the current state-of-the-art application whitelisting or contract-based security model (e.g. Android) by whitelisting a sub-Turing language class that can be formally reasoned about in a decidable fashion.

By increasing the ability for automated test and verification tools to detect implementation flaws, especially those in input parsers, Crema aims to improve software security by putting a tool into the hands of software developers to employ LangSec principles transparently.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 The Crema Programming Language

3.1.1 Motivation

As part of this research effort, an open-source programming language environment called Crema was prototyped to demonstrate the benefits of a computationally restricted environment. With such a language, many development tasks could be implemented with language-guaranteed termination and would thus greatly ease verification of security-critical program elements such as input recognizers and parsers.

As most input-handling programs are intended by their developers to be transducers, performing computations on input, and generating a resultant output, termination is not a roadblock to a majority of development projects. Clearly, there are a number of special exceptions to this pattern such as: operating system scheduling loops, event-handling loops, server listen loops, and read-evaluate-print loops that interact with a user for an undetermined period of time. There are a number of common programming tasks that notoriously produce vulnerabilities that would see security benefits in such an environment.

The Crema language targeted the LLVM tool-chain, so the LLVM IR resulting from a Crema program can be easily linked with any programs that can be compiled with LLVM. This allows developers to insert provably-secure code modules to existing code bases rather than needing to recreate the code for an application or service from the beginning. It is possible to develop the security-critical elements of a program separately, such as those routines that parse input in Crema and verify them with higher assurance. New software projects may be written entirely or partially in Crema to take advantage of the security benefits automatically provided.

The LLVM compiler framework is a tool-chain of modular components to analyze, optimize, compile, and execute programs via a standardized byte-code intermediate-representation (IR) [14]. Front-ends parse an input language, construct an abstract syntax tree (AST), and emit LLVM IR which then can leverage the existing optimization passes, a cross-platform JIT compiler, and static analysis tools to allow for rapid compiler development. Once an input program has been converted to the IR, there are a number of tools and libraries that can then be used to optimize the IR for faster execution or smaller memory footprints.

3.1.2 Theory

The programming environment presented here is based on the classic Turing machine; however, the transition function δ is limited in such a way that it cannot return to a previously-visited state. Due to this limitation, the modified Turing machine will always terminate after all states in δ have been exhausted. To verify δ, there must be an upper limit to the number of states being searched, thus a time limit is not needed to determine if the verification is complete and exhaustive.

Just-in-time (JIT) Function Inliner and Loop Unroller

The model described above is highly restrictive and translates into a programming language with limited practical use. These limitations make looping and recursive function calls impossible to represent, because a RET in the function or a branch in the loop condition must return to a previously-visited state. In order to retain the benefits of the model, while still allowing function and bounded-loop semantics, a JIT preprocessor is presented. This preprocessor is similar to the unrolling presented in Refs. [15].

In the transition function, a set of states is grouped into a *model function* named with a unique symbol. When the execution reaches this symbol, the model function is duplicated, inserted, and overwrites the unique function symbol. This process can be imagined as a mapping application of a duplicate model function across all the unique function symbols in the program. When the transition function is loaded into the modified Turing machine, each function call is inlined with a duplicate set of states and transitions representing the semantics of the function body.

A set of states can also be designated as a loop body and defined with a parametrized variable for the number of iterations to unroll. The start of each loop is denoted with a unique symbol. As the modified machine executes and the beginning of a loop symbol is reached, the JIT loop unroller interrupts the machine and creates n duplicate groups of the loop body. As a caveat, this unroller will optionally inline any functions within the loop as well. The upper bound on the number of iterations n must be known at the time the loop begins (Walther recursion [16]) and thus must be a function of the inputs read from the tape and constant values. These new duplicate states are now able to be reached at least once according to this machine specification.

With this JIT inliner and the loop unroller, our modified Turing machine can emulate many of the semantics found in popular general-purpose programming languages and their associated run-time environments. The exception, of course, is unbounded loops and loops where the number of iterations is not known a priori.

3.1.3 Crema Implementation

Crema Design Aspirations
Crema was designed to be a limited programming language with a minimal learning curve that provides a restricted but practical computational environment. It is a weakly-typed procedural language supporting implicit up-casting with syntax similar to Ruby or C with the exception of not allowing unbounded loops or non-terminating recursion (i.e. co-recursion).

The Crema grammar is shown in Bakus-Naur Form (BNF) in Figure 1. A unique feature of this grammar is the core looping construct is the '**`foreach`**' instruction, which iterates through a list and performs the desired computations on each element. As part of the Crema standard library, a **`crema_seq(start,end)`** function is provided to generate a sequence of consecutive numbers in a list for looping a certain number of times. While these constructs can be found in other programming languages (e.g. `seq()` in R, `foreach` in PHP and LISP), what is distinctive about Crema is that it does not support unbounded looping constructs such as `while` in C-like languages or `loop` in LISP-like languages.

An example FizzBuzz program written in Crema program is shown in Figure 2, where the words "Fizz", "Buzz", or "FizzBuzz" are printed depending on whether a number in the integer sequence from 1 to 100 is evenly divisible by 3, 5, or 15 [17]. This example serves to provide an impression of how most programs appear in the restricted environment.

The Crema compiler, *cremacc*, processes these input programs and converts them to LLVM IR byte-code, which can be optionally compiled to native machine code for the current platform. The abbreviated LLVM assembly that is produced from the FizzBuzz program is shown in Figure 3. If the program is kept in the IR format, it is easily portable to other LLVM-supported platforms and can be used as input for the existing tools that target the LLVM tool chain.

3.1.4 Crema Qmail Experiments with KLEE

After developing a functioning prototype of the Crema language and programming environment, a series of tests were performed to measure how effective the restricted computational model was in reducing the execution state space. These benchmarks were measured using the KLEE LLVM symbolic execution engine, which was held to the default three-minute time limit for execution. For the benchmark test, we used both C and Crema versions of the mail transport agent *qmail* because of its simplicity and practicality. Each test is described below, and the interpretation of the results continues in Section 4.0.

3.1.4.1 Qmail: a Resilient Secure Mail Transfer Agent (MTA)

D.J. Bernstein's qmail is an MTA designed with security as a core requirement [18]. In 1997, the Bernstein offered a reward to anyone who could report a security vulnerability in qmail, and the only time the bounty was claimed was in 2009. In describing the lessons learned during the ten years of the MTA's development, Bernstein highlights the dangers of parsing input. Accordingly, he worked to keep qmail's internal file formats as simple as possible. He recognized the SMTP parser as being the highest risk of compromise, and isolated it in a separate, untrusted process. Qmail's lessons remain highly relevant for today's high profile

input-related vulnerabilities such as Heartbleed, GnuTLS Hello overflow, MS SChannel, BERserk, and others.

To empirically measure the benefits of this restricted programming environment with respect to verification, the *qmail-smtpd* parser code was executed symbolically with KLEE while recording metrics of the state-space growth in two different testing scenarios. The qmail parser is an example of well-designed code written in a fully TC environment with security as a top priority. A similar parser was developed in the restricted Crema programming language in order to compare the code-coverage and statespace explosion when tested by KLEE.

```
<program> ::= <statements> | <empty>
<block> ::= '{' <statements> '}' | '{' '}'
<statements> ::= <statement> | <statements> <statement>
<statement> ::= <var_decl>
              | <struct_decl>
              | <func_decl>
              | <assignment>
              | <conditional>
              | <loop>
              | <return>
<var_decl> ::= <type> <identifier>
             | <type> <identifier> '=' <expression>
             | 'struct' <identifier> <identifier>
             | 'struct' <identifier> <identifier> '=' <identifier>
             | <list_decl>
<struct_decl> ::= 'struct' <identifier> '{' <var_decls> '}'
<func_decl> ::= <def> <type> <identifier> '(' <func_decl_arg_list> ')' <block>
              | <def> <type> '[' ']' <identifier> '(' <func_decl_arg_list> ')' <block>
              | 'extern' <def> <type> <identifier> '(' <func_decl_arg_list> ')'
              | 'extern' <def> <type> '[' ']' <identifier '(' <func_decl_arg_list> ')'
<assignment> ::= <identifier> '=' <expression>
               | <list_access> '=' <expression>
               | <struct> '=' <expression>
<conditional> ::= 'if' '(' <expression> ')' <block>
                | 'if' ')' <expression> ')' <block> 'else' <block>
                | 'if' '(' <expression> ')' <block> 'else' <conditional>
<loop> ::= 'foreach' '(' <identifier> 'as' <identifier> ')' <block>
<return> ::= 'return' <expression>
<type> ::= 'double' | 'int' | 'str' | 'void' | 'bool'
<var_decls> ::= <var_decl> | <var_decls> ',' <var_decl> | <empty>
<def> ::= 'def'
<func_decl_arg_list> ::= <var_decl>
                       | <func_decl_arg_list> ',' <var_decl>
                       | <empty>
<list_decl> ::= <type> <identifier> '{' '}'
              | <type> <identifier> '{' '}' '=' <expression>
<expression> ::= <term>
               | <expression> <bitwise> <term>
               | <expression> '+' <term>
               | <expression> '0' <term>
<term> ::= <factor>
         | <term> '*' <factor>
         | <term> '/' <factor>
         | <term> '%' <factor> | <term> <comparison> <factor>
<comparison> ::= '==' | '!=' | '>' | '=' | '<=' | '&&' | '||'
<bitwise> ::= '&' | '^' | '|'
<factor> ::= <var_access>
           | <list>
           | <value>
           | <identifier> '(' <func_call_arg_list> ')'
           | '(' <expression> ')'
           | '-' '(' <expression> ')'
<var_access ::= <identifier> | <list_access> | <struct> | '-' <var_access>
<list_access> ::= <identifier> '[' <expression> ']' | <identifier> '[' ']'
<struct> ::= <identifier> '.' <identifier>
<list> ::= '{' <func_call_arg_list> '}'
<func_call_arg_list> ::= <expression> | <func_call_arg_list> '.' <expression> | <empty>
<value> ::= <numeric> | <string> | 'true' | 'false'
<numeric> ::= <double> | '-' <double> | <int> | '-' <int
```

Figure 1: Crema Grammar in Bakus-Naur Form (BNF).

```
int hundred[] = crema_seq(1,100)

foreach(hundred as i){
  int_print(i)
  str_print(" ")
  if(i % 3 == 0) {
    str_print("Fizz")
  }
  if(i % 5 == 0) {
    str_print("Buzz")
  }
  str_println(" ")
}
```

Figure 2: Sample "FizzBuzz" Program Written in Crema

```
; ModuleID = 'Crema JIT'
target triple = "x86 64-pc-linux-gnu"

@hundred = internal global i8 * undef
@loopItCnter = internal global i64 undef
@i = internal global i64 undef

define i64 @main( i64 %argc , i8 ** %argv ) {
entry :
  call void @save args (
    i64 %argc , i8 ** %argv )
  %0 = call i8 * @crema seq ( i64 1 , i64 100)

  store i8 * %0, i8 ** @hundred
  br label %preblock
preblock :
  store i64 0 , i64* @loopItCnter
  br label %bodyblock
bodyblock :
  %1 = load i8 ** @hundred
  %2 = load i64* @loopItCnter
  %3 = call i64
    @int list retrieve ( i8 * \%1, i64 \%2)
  store i64 %3, i64* @i
  %4 = load i64* @i
  call void @int print ( i64 %4)
  %5 = call i8 * @str create ()
  call void @str append ( i8 * %5, i8 32)
  call void @str print ( i8 * %5)
  %6 = load i64* @i
  %7 = srem i64 %6, 3
  %8 = icmp eq i64 %7, 0
  br i1 %8, label %15, label %17
loopcondblock :
  %9 = load i64* @loopItCnter
  %10 = add i64 1 , %9
  store i64 %10, i64* @loopItCnter
  %11 = load i8 ** @hundred
  %12 = call i64 @list length ( i8 * %11)
  %13 = load i64* @loopItCnter
  %14 = icmp eq i64 %13, %12
  br i1 %14, label %termblock , label %bodyblock
; <label >:15
  %16 = call i8 * @str create ()
  call void @str append ( i8 * %16, i8 70)
  call void @str append ( i8 * %16, i8 105)
  call void @str append ( i8 * %16, i8 122)
  call void @str append ( i8 * %16, i8 122)
  call void @str print ( i8 * %16)
  br label %17
; <label >:17
  %18 = load i64* @i
  %19 = srem i64 %18, 5
  %20 = icmp eq i64 %19, 0
  br i1 %20, label %21, label %23
; <label >:21
  %22 = call i8 * @str create ()
  call void @str append ( i8 * %22, i8 66)
  call void @str append ( i8 * %22, i8 117)
  call void @str append ( i8 * %22, i8 122)
  call void @str append ( i8 * %22, i8 122)
  call void @str print ( i8 * %22)
  br label %23
; <label >:23
  %24 = call i8 * @str create ()
  call void @str append ( i8 * %24, i8 32)
  call void @str println ( i8 * %24)
  br label %loopcondblock
termblock :
  ret i64 0
}
```

Figure 3: LLVM Assembly Output of Crema FizzBuzz Program

3.1.4.2 KLEE: A Symbolic Virtual Machine

There are several tools designed to aid in the analysis and verification of input programs, one of which is KLEE [19], a tool to symbolically execute input programs to search for test-cases that will lead to an error or crash. KLEE attempts to exercise all possible branches of the input program's control flow graph by substituting a symbolic value for each branch condition and continuing down both branches. If a certain path causes an error or program crash, it will use a constraint-solver on the symbolic branch conditions leading to the error in order to generate a concrete input test case.

The scope of these conditions must be carefully limited by human-provided hints to prevent the solver from crashing or running for an unreasonable/unbounded amount of time. This is of greater importance in programs that contain unbounded loops as the KLEE engine considers a loop iteration to be a new state, thus running the verification endlessly. Due to the Halting Problem's undecidability for TC languages, KLEE cannot determine whether a program will terminate on a given input, thus it will execute the program symbolically for a certain time, which is three minutes by default and then terminate. Due to this time limit, there are some input programs for which KLEE cannot exercise the entirety of the control flow graph.

KLEE was used to explore state space growth of programs and detect crash cases, not to verify certain properties about the program, thus no source code annotations are used. Qmail was chosen for comparison, because its design stresses the perils of parsing and the necessity to isolate parsers of data coming from the network using all possible OS resources. This is consistent with the traditional view of LangSec and provides us with the parser cleanly separated from the rest of the MTA logic. Thus, using qmail removes the challenge of having to draw arbitrary boundaries between the code that validates inputs and the code where validation is assumed. (The code that validates inputs must be verified to recognize and validate the exact specified inputs.) With qmail, this separation is demanded explicitly and is approached closely, if not perfectly.

The second reason qmail was chosen is because its input-parsing tasks are clearly representative of the real challenges programs face, where untrusted input must be taken from the Internet. Despite being simple, SMTP has seen a number of notoriously unsafe implementations prior to qmail with notorious input-handling bugs. At the same time, an MTA for this widely-deployed protocol puts qmail under pressure to accommodate existing input variations and dialects rather than subset them and discard all non-compliant inputs within the chosen subset. Qmail's parsing is representative of the complexity associated with handling a broadly-deployed legacy protocol.

3.1.4.3 Test 1: Removal of qmail-stmpd Parser

In the first test, the qmail-smtpd parser is removed from the unbounded input loop, and the parser function is passed a symbolic input of increasing length. It is then symbolically executed by KLEE, as in the verification of a typical program or in an automatic exploit generation (AEG) tool's search. KLEE is instructed with `the --only-output-states-covering-new` option to keep separate statistics for the total number of paths explored and the number of unique paths without revisiting states.

This scenario is not guaranteed to exercise the entire CFG and may miss certain states due to the state-space explosion. The unique path statistic is used to model the restricted transition function described in Section 3.1.2. The difference in growth rates of total and trimmed paths was measured and is detailed in Section 4.0.

3.1.4.4 Test 2: Modifying qmail-smtpd to Crema-equivalent Code

The qmail-smtpd program has two significant functions that handle input parsing: `commands` and `addrparse`. The initial parsing routine (`commands`) is the simpler of the two. It reads a string from an input buffer and attempts to split the string into a command and an optional argument. The command portion of the input string parsed by `commands()` is used to index a table of function handlers. Specific commands will trigger the second parsing routine (`addrparse`) to parse the argument. For the sake of simplicity, the entire input was defined to be symbolic, but in order to target the second parsing routine the command portion was fixed using the `klee_assume()` function. The argument was at a maximum eight bytes long in order to maximize code coverage while restricting state space growth.

In the second test, the qmail-smtpd parser code is modified to simulate execution in a restricted computation environment equivalent to the Crema execution mode. Unbounded loops are removed, and tests are performed with fixed-length symbolic inputs to limit the number of iterations performed by input parsing loops. This approximates the transducer-like environment explained in Section 3.1.1. In this environment, the lack of unbounded loops drastically reduces the state space and verification is faster. This enables KLEE to practically handle larger code-bases for checking.

In this latter test, both the modified and unmodified qmail-smtpd parsers were symbolically executed multiple times with increasing time limits. The number of paths explored, instruction coverage, and average program states, among other metrics, were all recorded for each test. (Note: Many paths cannot be explored without specifying the length of the input string in qmail. This constraint makes run times explode and limits the experiment.) This test aims to show the benefits of programming in a language with restricted computational expressiveness, as described in Section 3.1.

3.1.4.5 Test 3: Testing an Input Crema Parser

This test used KLEE to exercise a qmail-like parser written in Crema. The goal of this test was to explore state-space growth in a restricted and sub-Turing programming environment versus a well-designed, but TC parser. The source code for the parser (edited slightly for brevity and clarity) can be found in Figure 4 along with the corresponding qmail commands parser functionality in Figure 5 with the infinite loop commented out. Both of these parsers were executed symbolically with KLEE to compare how a well-written and security-conscious parser written in C would compare with a prototype Crema parser *vis-a-vis* state-space growth. The results for this test are discussed in Section 4.0.

4.0 RESULTS AND DISCUSSION

The results of these three experiments are outlined below for both testing scenarios. It should be noted that due to the complexity of the C standard library, the instruction and branch code coverage of KLEE during execution is greatly diminished. As a control test for KLEE, the simple “Hello, World!” program was written in both C and Crema, and symbolically executed in KLEE. The resulting code coverage metrics can be found in Table 1.

```c
#include <stdio.h>

int main(int argc, char ** argv)
{
  printf("Hello World\n");
  return 0;
}
```

Figure 4: Hello World program written in C

```
str_println("Hello World")
```

Figure 5: Hello World program written in Crema

4.1 Test 1 Results

The results comparing the difference in growth between the total paths and trimmed paths are shown in Table 2. The length of the symbolic input string used as an input parameter for the qmail parser was varied to exercise an increasing amount of the code base. The execution times for inputs greater than nine characters were too long to return results. The percentage of total code exercised is shown in the table, as well as the total run-time for the KLEE symbolic engine. The number of paths explored quantify the growth in the state-space that the KLEE engine must explore to fully exercise the code reachable based on the symbolic input format. A comparison between the explored paths and trimmed paths as a function of symbolic input length is shown in Figure 8.

4.2 Test 2 Results

The results from Test 2, shown in Table 3 highlight the verification benefits of restricting the computational expressiveness of the program source code, a goal of the language described in Section 3.1. This test shows the improved code coverage and decreased number of states that must be checked when compared with a standard implementation. These results more closely mirror the program after the function inliner and loop unroller described in Section 3.1.2. In Figure 9, a plot provides the number of paths the KLEE symbolic engine was able to explore in one minute before it was halted. In this test case, the number of paths that KLEE was able to explore (i.e. instruction coverage) was nearly one order of magnitude greater for the bounded parser than the unbounded parser.

4.3 Test 3 Results

The results from Test 3, shown in Table 4 highlight the state-space size growth advantages of the restricted computational environment that Crema has to offer. The number of states that KLEE explored is significantly reduced in the Crema parser when compared to similar C code from

qmail. This empirically reaffirms what is intuitively expected: Verification and exercising restricted computational models is easier than fully-TC environments. Comparing the code coverage between the two qmail parsers shows that the Crema program covers a considerably higher percentage of instructions. It is worth remarking that KLEE's ability to cover a large portion of the code is limited even for simple the simple Hello World programs.

```
1.  def int commands(string c){
2.    string cmd
3.    string arg
4.    int len = str_len(c)
5.    int itr = 0
6.    int max_input_itr[] = crema_seq(0, len)
7.    foreach(max_input_itr as itr) {
8.      if (c[itr] == '\n') {
9.        if (itr > 0) {
10.         int i = itr - 1
11.         cmd = str_substr(c, 0, i)
12.     break
13.       }
14.     }
15.   }
16.   len = str_len(c)
17.   itr = str_chr(cmd, ' ')
18.   arg = str_substr(cmd, itr, 0)
19.   if (str_compare(cmd, "rcpt") == 1) {
20.     smtp_rcpt(arg)
21.   } else if (str_compare(cmd, "mail") == 1) {
22.       smtp_mail(arg)
23.   } else if (str_compare(cmd, "data") == 1) {
24.       smtp_data()
25.   } else if (str_compare(cmd, "quit") == 1) {
26.       smtp_quit()
27.   } else if (str_compare(cmd, "helo") == 1) {
28.       smtp_helo(arg)
29.   } else if (str_compare(cmd, "ehlo") == 1) {
30.       smtp_ehlo(arg)
31.   } else if (str_compare(cmd, "rset") == 1) {
32.       smtp_rset()
33.   } else if (str_compare(cmd, "help") == 1) {
34.       smtp_help()
35.   } else if (str_compare(cmd, "noop") == 1) {
36.       err_noop()
37.   } else if (str_compare(cmd, "vrfy") == 1) {
38.       err_vrfy()
39.   } else {
40.       err_unimpl()
41.   }
42.   return 0
43. }
44. int argc = prog_arg_count()
45. string command = prog_argument(1)
46. commands(command)
```

Figure 6: Source Code for Crema Parser Prototype

```
int commands(ss,c)
char *ss;
struct commands *c;
{
  int i;
  char *arg;

//for (;;) {
    if (!stralloc copys(&cmd,""))
        return -1;
    for (;;) {
      if (!stralloc_readyplus(&cmd,1))
        return -1;
      cmd.s[cmd.len] = ss[cmd.len];
      if (cmd.s[cmd.len] == '\0')
        return 0;
      if (cmd.s[cmd.len] == '\n')
        break;
      ++cmd.len;
    }
    if (cmd.len > 0)
        if (cmd.s[cmd.len - 1] == '\r')
            --cmd.len;
    cmd.s[cmd.len] = 0;
    i = str_chr(cmd.s,' ');
    arg = cmd.s + i;
    while (*arg == ' ') ++arg;
    cmd.s[i] = 0;

    for (i = 0;c[i].text;++i)
        if (case_equals(c[i].text,cmd.s))
            break;
    c[i].fun(arg);
    if (c[i].flush) c[i].flush();
//}
}
struct commands smtpcommands[] = {
    { "rcpt", smtp_rcpt, 0 }
    , { "mail", smtp_mail, 0 }
    , { "data", smtp_data, flush }
    , { "quit", smtp_quit, flush }
    , { "helo", smtp_helo, flush }
    , { "ehlo", smtp_ehlo, flush }
    , { "rset", smtp_rset, 0 }
    , { "help", smtp_help, flush }
    , { "noop", err_noop, flush }
    , { "vrfy", err_vrfy, flush }
    , { 0, err unimpl, flush }
};

int main(int argc, char ** argv)
{
    if (commands(argv[1], &smtpcommands) == 0)
      die_read();
    die_nomem();
}
```

Figure 7: Minimal Qmail Parser Source Code

Table 1: KLEE Coverage for "Hello, World" Program

Input Source Language	Instruction Coverage (%)	Branch Coverage (%)
C	17.10	11.56
Crema	30.10	17.89

Table 2: Qmail State-space Explosion Reduction in Restricted Environment

Symbolic Input Size (characters)	Instruction Coverage (%)	Explored Paths	Trimmed KLEE Paths	KLEE Run-time (s)
5	31.37	997	12	1.12
6	31.59	1780	15	2.03
7	33.13	2985	24	6.23
8	33.97	4737	33	53.27
9	34.62	7731	45	540.2

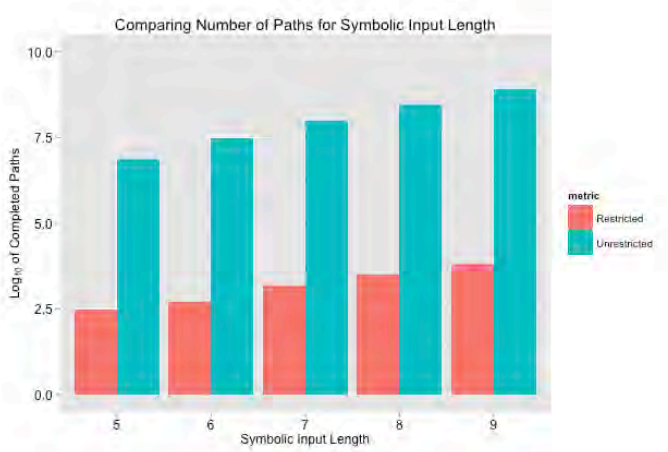

Figure 8: Explored Paths and Trimmed Paths vs. Symbolic Input Length

Table 3: Completed Paths in Time Limit for Bounded and Unbounded Parser

Time Limit (s)	Completed Paths in Unbounded Parser	Completed Paths in Bounded Parser
2	1068	551
3	1412	847
4	1803	880
5	2115	966
6	2573	1246
7	2921	1295
8	3263	1283
9	3413	1344
Time Limit (s)	**Completed Paths in Unbounded Parser**	**Completed Paths in Bounded Parser**
10	3977	1363
20	6679	1613
30	8365	1925
40	9899	2386
60	12451	2783

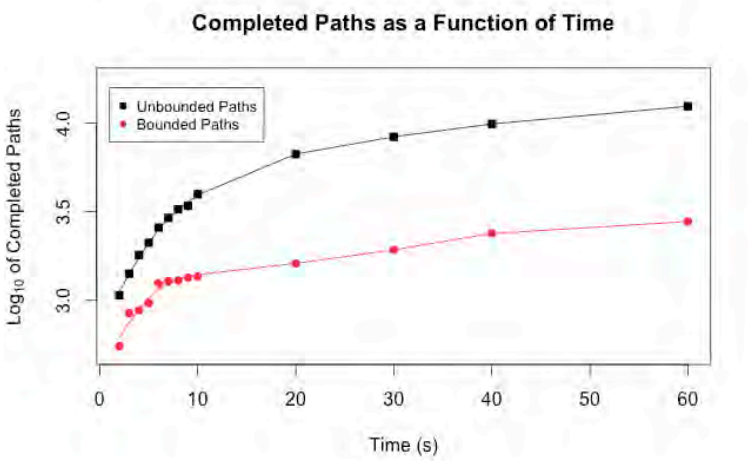

Figure 9: Comparison of Restricted and Normal Symbolic Executions

Table 4: Qmail C Parser Compared to Crema Parser

Parser	Execution Time (s)	Maximum States	Instruction Coverage (%)	Branch Coverage (%)
qmail C	31.50	678	44.47	33.96
Crema	28.67	76	61.97	37.74

4.4 Interpretation of Results

As shown, the restricted environment results in a clear reduction of the state-space. Experience suggests that this reduction is significant. Recently, NICTA formally verified the seL4 microkernel [20], whose 10,000 lines of high-level code translate approximately into 10,000 lines of automatic and provable C code. Larger code-bases have resisted verification due to the state-space explosion. The results from our experiment suggest as much as an entire order of magnitude reduction in the state-space during verification. We note that this reduction occurred in a well-designed but unmodified code environment with security in mind. Code that specifically targets such reductions is likely to make a more significant impact on the state-space.

Many components of existing, unverified software projects can be modeled in such a restricted fashion and then verified. For example, components of the Linux kernel could be modeled in a restricted environment to avoid synchronization and critical sections and then verified for correctness improving the assurances provided by the platform.

4.5 Related Works

4.5.1 Boogie Programming Language

Crema falls into a class of languages commonly referred to as intermediate verification languages. And while the idea of setting finite boundaries on recursive and looping functions is not new, Crema offers a more practical and modular approach with its LLVM backend.

An approach similar to that of Crema's is the Corral program verifier [21] for the Boogie intermediate verification language from Microsoft [22]. The Corral whole-program analyzer employs recursion bounding with a technique called *stratified inlining*. This involves a set of assumptions about the program's behavior and uses approximation methods to locate assertion errors. Although Crema and Corral share the same goal and employ function inlining for loops and recursion, Corral and Boogie coupled together do not operate within the bounds of a sub-Turing grammar. Finally, Corral analyzes entire programs and uses the Z3 theorem solver to check for bugs. Crema on the other hand provides a much more flexible platform for its bug checking or symbolic execution—the emitted LLVM assembly code can be applied to several existing automated software testing suites.

4.5.2 Modeling Execution Events as an Input Language

Reference [23] defines a reference monitor as automaton that recognizes a language of events. This model interprets the events and actions from a program as a stream of symbols. It can reject incorrect input, which is used to detect compromises in monitored processes. It can also evaluate whether a process is starting to misbehave based on certain patterns of events; however, it cannot recognize more complex input language grammar classes. The monitor is restricted to only checking input prefixes for a general process due to the undecidability of the Halting Problem.

With the restricted model like Crema, a reference monitor could potentially recognize more complex languages of inputs and possibly roll back events performed by a compromised process after it terminated, thus recovering a trustworthy state. Presently, a reference monitor is limited by two factors: the undecidability cliff and operating at the same level of computational expressiveness as the process it is monitoring. Similar to the cat-and-mouse game with malware and anti-viruses operating at the same level of privilege, the fact that the reference monitor is not more powerful than the monitored process weakens its abilities to recognize compromise.

4.5.3 Automated Exploit Generation

AEG is the effort to automate the hunt for vulnerabilities and provide conceptual proofs of their exploits [24]. A typical example of this would be the examination of inputs that cause a computation to be diverted in such a way that an attacker gains full control of a target.

AEG started by automating the process of crafting input programs or payloads for the classic execution model of stack buffer overflows as described in Reference [1]. This model is nearly extinct in modern desktop software because of defensive measures like data execution prevention, address space layout randomization, and Microsoft's Enhance Mitigation Experience Toolkit. Since the 1990s, these defensive techniques co-evolved with the state-of-the-art offensive methods, and this model thrives on the ubiquitous micro-controller firmware that appears poised to drive the so-called "Internet of Things." In a recent publication, the authors of Reference [25] described AEG as a program verification task but with a twist. Typical safety properties are replaced with finding an unexpected program execution path when subjected to crafted inputs. From a LangSec perspective, an AEG algorithm actually finds both a description

of an input-driven weird machine as well as the malicious exploit that drives it to some definition of an undeniably unexpected computation.
The verification and AEG research areas are closely related, impacts to one field directly affect the other [26]. Verification aims to prove that there are no unintended states in the program that are reachable, which is often compared to a formal model. For AEG, the generator is attempting to prove that there exists a reachable, unintended error state. As stated in Reference [25].

> *"Casting AEG in a verification framework ensures AEG techniques are based on a firm theoretic foundation. The verification-based approach guarantees sound analysis, and automatically generating an exploit provides proof that the reported bug is security-critical."*

Both verification and AEG are impacted by state-space explosion when mapping the CFG for TC software programs. Additionally, the Halting Problem introduces the issue of determining whether to continue searching or stop.

The problem of automating verification and AEG is exacerbated by computational complexity of general-purpose programming environments. The general-purpose programming languages today provide more computational expressiveness than is needed to perform most input-validation software tasks. The power needed to perform a particular computation must always be less than the power provided by the programming language. Maintaining a narrow gap between the two is essential to prevent inherent security risks [27]. This gap, known as the *undecidability cliff*, prevents both formal verification and AEG tools from analyzing an entire general, non-trivial input program [28].

5.0 CONCLUSION

For most inputs, the fully expressive, TC environment is overly powerful and carries a significant and realized risk of compromise. The majority of programming languages aims for Turing-completeness, then focuses on syntax and library functions. The limited model and the Crema language described in this paper have been designed to not aim for Turing-completeness, but rather with a view towards practically and safety. This allows program-verification tools to explore the state space of programs. By providing a restricted execution model and showing the verification benefits, future work can explore existing code bases and perform analyses that identify components or sub-systems that could be modeled in this restricted environment. The computational model and its corresponding language, Crema, are the first steps in bringing a formal verification to code bases that were once deemed too large for symbolic execution.

Future Crema research and development will take this restricted model and move it closer to the hardware, either through field-programmable gate arrays or real-time operating system support for Crema-like languages. Using the forward-only execution environment as describe in this report, we forecast that this type of model would work well in a system similar to Google's NaCl environment. This would allow users to run binaries in a safe environment.

In this initial effort, it was shown that state space growth was reduced when traditional methods were applied. Now that the language is less expressive, we would like to explore more powerful

formal methods in the future and realize techniques that were once thought to be impossible. This would involve analysis on a program and developing a system that prompts the user to write a section of their code in Crema when formal analysis becomes too difficult.

Another powerful feature Crema could potentially offer is to automatically convert TC code to sub-TC code. Although doing this in a generalized way is not possible, regions of code could be identified automatically. Our vision the future of secure programming is one in which input-driven computations are programmed with parsers that work with the programmer to improve the security of their code.

6.0 REFERENCES

[1] Aleph One, "Smashing the Stack for Fun and Profit," *Phrack*, pp. Vol. 7, 49:14, 11 August 1996.

[2] S. Bratus, T. Darley, M. Locasto, M. L. Patterson, R. Shapiro and A. Shubina, "Beyond Planted Bugs in "Trusting Trust": The Input-Processing Frontier," *IEEE Security & Privacy*, pp. 83-87, January/February 2014.

[3] S. Bratus, M. Locasto, M. Patterson, L. Sassaman and A. Shubina, "Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation," *;login:*, vol. 36, no. 6, pp. 13-21, Dec. 2011.

[4] L. Sassaman, M. Patterson, S. Bratus and M. Locasto, "Security applications of formal language theory," *IEEE Systems Journal*, pp. 489-500, 2013.

[5] E. Nava and D. Lindsay, "Abusing internet explorer 8's xss filters," Apr. 2010. [Online]. Available: http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf. [Accessed 12 Feb. 2015].

[6] S. Bratus, M. Locasto, M. Patterson, L. Sassaman and A. Shubina, "Exploit Programming: From Buffer Overflows to 'Weird Machines' and Theory of Computation," *;login*, pp. 13-21, December 2011.

[7] J. Vanegue, "The weird machines in proof-carrying code," in *Proc. First Annual Langsec Workshop*, 2014.

[8] Nergal, "The advanced return-into-lib(c) exploits," Dec. 2001. [Online]. Available: http://phrack.org/issues/58/4.html.

[9] G. Richarte, "Re: Future of buffer overflows," Oct. 2000. [Online]. Available: http://seclists.org/bugtraq/2000/Nov/32.

[10] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[11] R. Hund, T. Holz and F. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proceedings of the 18th Conference on USENIX Security Symposium*, Berkeley, CA, 2009.

[12] L. Sassaman, M. Patterson, S. Bratus, M. Loscato and A. Shubina, "Security Applications of Formal Language Theory," Dartmouth College, Hanover, NH, 2011.

[13] J. Mitolla, III, "Software radio architecture: a mathematical perspective," *IEEE Journal on*

Selected Areas in Communications , pp. 514-538, 1999.

[14] C. Lattner, "The llvm compiler infrastructure," 2015. [Online]. Available: http://llvm.org. [Accessed 20 Feb. 2015].

[15] A. Lal, S. Qadeer and S. Lahiri, "Corral: A whole-program analyzer for boogie".

[16] C. Walther, "Security applications of formal language theory," *Artificial Intelligence,* vol. 70, no. 1, 1994.

[17] I. Ghory, "Using fizzbuzz to find developers who grok coding," Jan. 2007. [Online]. Available: http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/. [Accessed 20 Feb. 2015].

[18] D. Bernstein, "qmail," 2013. [Online]. Available: http://cr.yp.to/qmail.html.

[19] C. Cadar, D. Dunbar and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of USENIX OSDI 2008,* San Diego, CA, 2008.

[20] G. Klein, "Operating system verification - an overview," *Sadhana,* vol. 34, no. 1, pp. 27-69, 2009.

[21] A. Lal, S. Qadeer and S. Lahiri, "Corral: A Whole-Program Analyzer for Boogie," in *First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, 2011.

[22] Microsoft Corporation, "Microsoft Research Boogie," 22 Oct. 2012. [Online]. Available: https://boogie.codeplex.com/. [Accessed 26 Feb. 2015].

[23] F. Schneider, "Enforceable Security Policies," *ACM Transactions on Information and System Security,* pp. 30-50, February 2000.

[24] S. Heelan, *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities,* Oxford, UK: Master's thesis, University of Oxford, 2009.

[25] T. Avgerinos, "Automatic Exploit Generation," *Communications of the ACM,* vol. 57, no. 2, pp. 74-84, 2014.

[26] J. Vanegue, *The automated exploitation grand challenge,* H2HC, 2013.

[27] L. Sassaman, M. Patterson, S. Bratus and A. Shubina, "The halting problem of network stack insecurity," *USENIX ;login:,* vol. 36, no. 6, pp. 22-32, 2011.

[28] S. Bratus and F. Linder, "Information security war room," in *Proc. USENIX,* 2014.

[29] B. Cook, A. Podelski and A. Rybalchenko, "Termination proofs for system code," in *Proceedings of the 2006 ACM SIGPLAN conference,* New York, 2006.

[30] S. T. Taft and F. Olsen, "Ada helps churn out less-buggy code," *Government Computer News,* pp. 2-3, June 1999.

[31] D. Turner, "Total Functional Programming," *Journal of Universal Computer Science. Vol. 10, No. 7.*, pp. 751-768, 2004.

[32] P. Wadler, "Comprehending Monads," in *Proceedings of the 1990 ACM conference on LISP and functional programming,* Nice, France, 1990.

[33] E. Bosman and H. Bos, "Framing Signals - A Return to Portable Shellcode," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14),* Washington, DC, 2014.

[34] U. Schoning, Theoretische Informatik - kurz gefaast. 5th ed., Heidelberg, Germany: Spektrum, 2008.

APPENDIX – CREMA FUNCTIONALITY

The state of the Crema compiler at the end of this effort contains many of the basic functions one would expect to find in a general-purpose programming language. This section offers a list of Crema's current keywords and functionality.

A.1 Crema Keywords

Keyword	Definition
`as`	define loop iteration condition
`bool`	Boolean data type
`break`	passes control outside of conditional statement
`char`	character data type
`def`	function definition
`double`	floating point data type
`else`	completes conditional statement
`extern`	extend scope of a variable
`false`	Boolean value 'false'
`foreach`	iterates over a loop structure
`if`	begins conditional statement
`int`	Integer data type
`return`	return value
`sdef`	secure function definition
`string`	string data type
`struct`	data structure
`true`	Boolean value 'true'
`uint`	unsigned integer
`void`	defines no return value

A.2 Syntax Examples

The first example in Figure 10 shows basic arithmetic functions in Crema. Two integer variables *a* and *b* are assigned an initial value, and a *result* variable is declared. In order the operations shown are: addition, subtraction, multiplication, division, modulus, logical OR, logical AND, and logical XOR. These same operations can also be performed using the floating point and unsigned integer data types.

```
1.  int a = 2
2.  int b = 5
3.  int result
4.  result = a + b
5.  result = a - b
6.  result = a * b
7.  result = a / b
8.  result = a % b
9.  result = a | b
10. result = a & b
11. result = a ^ b
```

Figure 10: Basic Mathematical Operations in Crema

Figure 11 shows a simple function definition in Crema followed by a call to that function. The keyword def is used to begin the definition of a function followed by its return type. The function name is followed by a set of parenthesis that contains the functions arguments and the types associated with them.

```
1.  def int foo(int a)
2.  {
3.      return a + 32
4.  }
5.  int bar = foo(123)
```

Figure 11: Crema function definition and function call

The series of statements in Figure 12 are an example of how if-else statements are implemented in Crema. Unlike Python, Crema is not sensitive to the tabulation of the syntax, and the whitespaces in this example are not necessary for the Crema compiler to successfully interpret the if-else statement shown here. For multiple conditions, the `else` keyword may be replaced with `else if` followed by another conditional expression in parenthesis.

```
1.  int x=0
2.  int y=1
3.  int z
4.  if (x==0) {
5.    z=2
6.  } else {
7.    z=3
8.  }
```

Figure 12: Crema if-else statement

Line 1 in Figure 13 shows how array data structures are declared in Crema. Lines 2-8 provide an example of how to iterate over an array and that loops can be nested (Line 5). In its current state, Crema loop iterators are defined as integers. In other words, loops access array data by index.

```
1.  int ints[] = [1,2,3]
2.  foreach (ints as i)
3.  {
4.    int a = i
5.    foreach (ints as j)
6.    {
7.      int a = j
8.    }
9.  }
```

Figure 13: Array declaration and nested loops

Lines 1-4 in Figure 14 declare the struct data type named `bar`, which contains two integer variables. Line 5 then declares a second struct name `foo`, which is identical to `bar`. The `a` variable in `foo` is then assigned a value 34, and a new integer `x` is assigned the value of `foo.a` – 2, which is 32.

```
1.  struct bar {
2.    int a
3.    int b
4.  }
5.  struct bar foo
6.  foo.a = 34
7.  int x = foo.a - 2
```

Figure 14: Struct declaration with variable assignment

Crema contains a standard library that allows the user to perform many basic operations in addition to the ones previously described. The next three figures show all the basic functionality the standard library has to offer. This includes list and array manipulation, more advanced mathematical operations such as trigonometric functions, and basic print functions.

```
1.  list_t * list_create(int64_t es);
2.  void list_free(list_t * list);
3.  void list_insert(list_t * list, unsigned int idx, void * elem);
4.  void * list_retrieve(list_t * list, unsigned int idx);
5.  void list_append(list_t * list, void * elem);
6.  void list_concat(list_t * list1, list_t * list2);
7.  void list_delete(list_t * list, unsigned int idx);
8.  int64 t list length(list t * list);
9.  list_t * int_list_create();
10. void int list insert(list t * list, int64 t idx, int64 t val);
11. int64_t int_list_retrieve(list_t * list, int64_t idx);
12. void int_list_append(list_t * list, int64_t elem);
13. list_t * double_list_create();
14. void double_list_insert(list_t * list, unsigned int idx, double val);
15. double double_list_retrieve(list_t * list, unsigned int idx);
16. void double_list_append(list_t * list, double elem);
17. void double_print(double val);
18. void double_println(double val);
19. list_t * crema_seq(int64_t start, int64_t end);
```

Figure 15: List functions from the Crema standard library

```
1.  double double_floor(double val);
2.  double double_ceiling(double val);
3.  double double_round(double val);
4.  double double_truncate(double val);
5.  double double_square(double val);
6.  int64_t int_square(int64_t val);
7.  double double_pow(double base, double power);
8.  int64_t int_pow(int64_t base, int64_t power);
9.  double double_sin(double val);
10. double double_cos(double val);
11. double double_tan(double val);
12. double double_sqrt(double val);
13. double double_abs(double val);
14. int64_t int_abs(int64_t val);
```

Figure 16: Basic mathematical functions in Crema standard library

```
1.  string t * str create();
2.  void str_free(string_t * str);
3.  void str_insert(string_t * str, unsigned int idx, char elem);
4.  char str_retrieve(string_t * str, unsigned int idx);
5.  void str append(string t * str, char elem);
6.  void str_concat(string_t * str1, string_t * str2);
7.  void str_print(string_t * str);
8.  void str_println(string_t * str);
9.  void str delete(string t * str, unsigned int idx);
10. string_t * str_substr(string_t * str, unsigned int start, unsigned int len);
11. void double_print(double val);
12. void double_println(double val);
13. void int print(int64 t val);
14. void int println(int64 t val);
15.
16. // Type Conversion
17. int64 t double to int(double val);
18. double int_to_double(int64_t val);
19. string_t * int_to_string(int64_t val);
20. int64_t string_to_int(string_t * str);
21. int64_t string_to_double(string_t * str);
```

Figure 17: Basic print and type conversion functions in Crema standard library

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

Abbreviation	Full Name
AST	Abstract Syntax Tree
AEG	Automatic exploit generation
BNF	Bakus-Naur Form
CFG	Control flow graph
IR	Intermediate representation
JIT	Just-In-Time
LLVM	Low Level Virtual Machine
MTA	Mail transport agent
ROP	Return-oriented programming
stdlib	standard library
TC	Turing complete